# TrickBot gang template-based metaprogramming Bazar malware

By Kevin Henson

Published: 2022-02-02 · Archived: 2026-04-05 17:36:18 UTC

Kevin Henson

Malware Reverse Engineer

IBM

Malware authors use various techniques to obfuscate their code and protect against reverse engineering. Techniques such as control flow obfuscation using Obfuscator-LLVM and encryption are often observed in malware samples.

This post describes a specific technique that involves what is known as metaprogramming, or more specifically template-based metaprogramming, with a particular focus on its implementation in the Bazar family of malware (BazarBackdoor/BazarLoader). Bazar is best known for its ties to the cybercrime gang that develops and uses the TrickBot Trojan. It is a major cybercrime syndicate that is highly active in the online crime arena.

## A few words about metaprogramming

Metaprogramming is a technique where programs are designed to analyze or generate new code at runtime. Developers typically use metaprogramming techniques to make their code more efficient, modular and maintainable. Template-based metaprogramming incorporates templates that serve as models for code reuse. The templates can be written to handle multiple data types.

For example, the basic function template shown below can be used to define multiple functions that return the maximum of two values such as two numbers or two strings. The type is generalized in the template parameter <typename T>, as a result, a and b will be defined based on the usage of the function. One of the “magical” attributes of templates is that the max() function doesn’t actually exist until it’s called and compiled. For the example below, three functions will be created at compile time, one for each call.

```
//Sample function template
 template<typename T>
 T max (T a, T b)
 {
     // if b < a then yield a else yield b
    return b < a ? a : b;
 }

 // Calls to max()
 max(10,5);
```

```
max(5.5, 8.9);
max("reverse", "engineering");
```

Templates can be quite complex; however, this high-level understanding will suffice in grasping how the concept is used to a malware author's advantage.

## The latest tech news, backed by expert insights



## Malware development

Malware authors take advantage of the metaprogramming technique to both obfuscate important data and ensure that certain elements, such as code patterns and encryption keys, are generated uniquely with each compilation. This hinders analysis and makes developing signatures for static detection more difficult because the encryption code changes with each compiled sample.

The key components in metaprogramming used to accomplish this type of obfuscation are the templates and another feature called constexpr functions. In simple terms, a constexpr function's return value is determined at compile time.

To illustrate how this works, the following sections will compare samples compiled from the open-source library ADVobfuscator to Bazar samples found in the wild. The adoption of more advanced programming techniques within the Bazar malware family is especially relevant since the operators of Bazar are highly active in attacks against organizations across the globe.

### ADVobfuscator

To get a better understanding of how template programming is utilized with respect to string obfuscation, let's take a look at two header files from ADVobfuscator. ADVobfuscator is described as an "Obfuscation library based on C++11/14 and metaprogramming." The MetaRandom.h and MetaString.h header files from the library are discussed below.

### MetaRandom.h

The MetaRandom.h header file generates a pseudo-random number at compile time. The file implements the keyword constexpr in its template classes. The constexpr keyword declares that the value of a function or variable can be evaluated at compile time and, in this example, facilitates the generation of a pseudo-random integer seed based on the compilation time that is then used to generate a key.

```
namespace
 {
    // I use current (compile time) as a seed
    constexpr char time[] = __TIME__; // __TIME__ has the following format: hh:mm:ss in 24-hour tim
```

```
    // Convert time string (hh:mm:ss) into a number
    constexpr int DigitToInt(char c) { return c - '0'; }
    const int seed = DigitToInt(time[7]) +
                     DigitToInt(time[6]) * 10 +
                     DigitToInt(time[4]) * 60 +
                     DigitToInt(time[3]) * 600 +
                     DigitToInt(time[1]) * 3600 +
                     DigitToInt(time[0]) * 36000;
}
```

*Figure 1: Code Block 1 MetaRandom.h*

### MetaString.h

The MetaString.h header file consists of versions of a template class named *MetaString* that represents an encrypted string. Through template programming, MetaString can encrypt each string with a new algorithm and key during compilation of the code. As a result, a sample could be produced with the following string obfuscation:

- Each character in the string is XOR encrypted with the same key.
- Each character in the string is XOR encrypted with an incrementing key.
- The key is added to each character of the string. As a result, decryption requires subtracting the key from each character.

Here is a sample MetaString implementation from ADVobfuscator.

This template defines a MetaString with an algorithm number (N), a key value and a list of indexes. The algorithm number controls which of the three obfuscation methods are used and is determined at compile time.

```
template<int N, char Key, typename Indexes>
struct MetaString;
```

*Figure 2: Code Block 2 MetaString.h*

This is a specific implementation of MetaString based on the above template. The algorithm number (N) is 0, K is the pseudo-random key and I (Indexes) represent the character index in the string. When the algorithm number 0 is generated at compile time, this implementation is used to obfuscate the string. If the algorithm number 1 is generated, the corresponding implementation is used. *ADVobfuscator* uses the C++ macro *__COUNTER__* to generate the algorithm number.

```
template<char K, int… I>
 struct MetaString<0, K, Indexes<I…>>
 {
      // Constructor. Evaluated at compile time.
      constexpr ALWAYS_INLINE MetaString(const char* str)
```

```
        : key_{ K }, buffer_{ encrypt(str[I], K)… } { }

    // Runtime decryption. Most of the time, inlined
    inline const char* decrypt()
    {
        for (size_t i = 0; i < sizeof…(I); ++i)
            buffer_[i] = decrypt(buffer_[i]);
        buffer_[sizeof…(I)] = 0;
        LOG("- Implementation #" << 0 << " with key 0x" << hex(key_));
        return const_cast<const char*>(buffer_);
    }

 private:
    // Encrypt / decrypt a character of the original string with the key
    constexpr char key() const { return key_; }
    constexpr char ALWAYS_INLINE encrypt(char c, int k) const { return c ^ k; }
    constexpr char decrypt(char c) const { return encrypt(c, key()); }

    volatile int key_; // key. "volatile" is important to avoid uncontrolled over-optimization by
    volatile char buffer_[sizeof…(I) + 1]; // Buffer to store the encrypted string + terminating
};
```

*Figure 3: Code Block 3 MetaString.h*

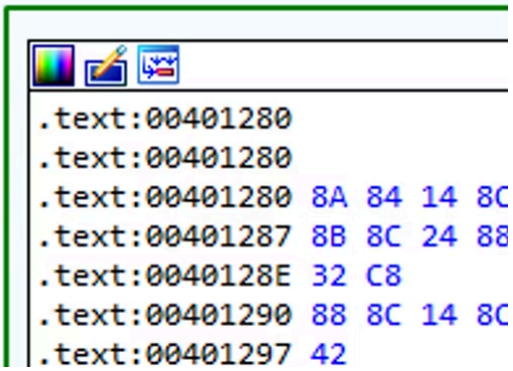
## ADVobfuscator samples

Interesting code patterns are observed when samples are built using ADVobfuscator. For example, after compiling the Visual Studio project found in the public Github repo, the resulting code shows the characters of the string being moved to the stack, followed by a decryption loop.

These snippets illustrate the dynamic nature of the library. Each string is obfuscated using one of the three obfuscation methods previously described. Not only are the methods different, the opcodes — the values in blue, which are commonly used in developing YARA rules — can vary as well for the same obfuscation method. This makes developing signatures, parsers and decoders more difficult for analysts. Notably, the same patterns are observed in *BazarLoader* and *BazarBackdoor* samples.

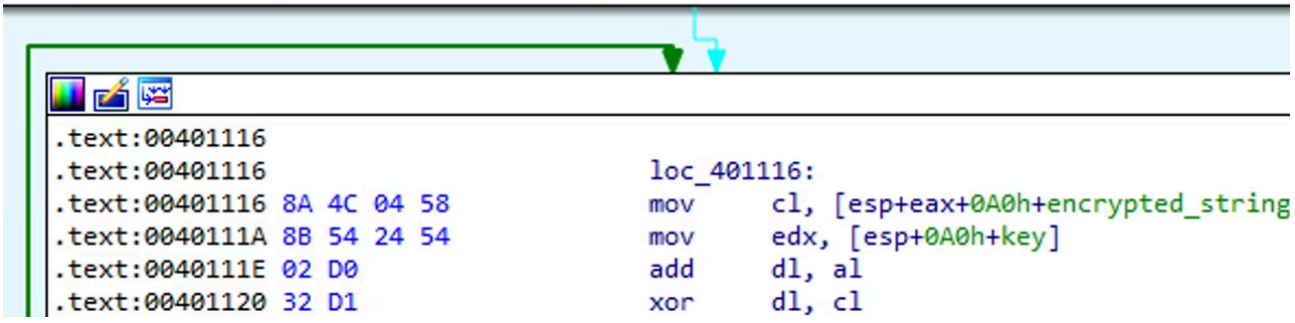
```
XOR encryption with the same key
```

```
C7 84 24 88 00 00 00 45+mov     [esp+0A0h+key_1], 45h ; 'E'
00 00 00
33 D2                   xor     edx, edx
C6 84 24 8C 00 00 00 07 mov     [esp+0A0h+encrypted_string], 7
C6 84 24 8D 00 00 00 37 mov     [esp+0A0h+var_13], 37h ; '7'
C6 84 24 8E 00 00 00 2C mov     [esp+0A0h+var_12], 2Ch ; ','
C6 84 24 8F 00 00 00 31 mov     [esp+0A0h+var_11], 31h ; '1'
C6 84 24 90 00 00 00 2B mov     [esp+0A0h+var_10], 2Bh ; '+'
C6 84 24 91 00 00 00 20 mov     [esp+0A0h+var_F], 20h ; ' '
C6 84 24 92 00 00 00 3C mov     [esp+0A0h+var_E], 3Ch ; '<'
C6 84 24 93 00 00 00 65 mov     [esp+0A0h+var_D], 65h ; 'e'
C6 84 24 94 00 00 00 16 mov     [esp+0A0h+var_C], 16h
C6 84 24 95 00 00 00 35 mov     [esp+0A0h+encrypted_string_1], 35h ;
C6 84 24 96 00 00 00 20 mov     [esp+0A0h+encrypted_string_1+1], 20h
C6 84 24 97 00 00 00 24 mov     [esp+0A0h+encrypted_string_1+2], 24h
C6 84 24 98 00 00 00 37 mov     [esp+0A0h+encrypted_string_1+3], 37h
C6 84 24 99 00 00 00 36 mov     [esp+0A0h+encrypted_string_1+4], 36h
8A 84 24 8C 00 00 00    mov     al, [esp+0A0h+encrypted_string]
C6 84 24 9A 00 00 00 00 mov     [esp+0A0h+encrypted_string_1+5], 0
0F 1F 40 00             nop     dword ptr [eax+00h]
66 0F 1F 84 00 00 00 00+nop     word ptr [eax+eax+00000000h]
00
```




XOR encryption with an incrementing key

```
text:004010BB 04 07                    add     al, 7
text:004010BD 34 20                    xor     al, 20h
text:004010BF 88 44 24 5F              mov     [esp+0A0h+var_41], al
text:004010C3 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:004010C7 04 08                    add     al, 8
text:004010C9 34 53                    xor     al, 53h
text:004010CB 88 44 24 60              mov     [esp+0A0h+var_40], al
text:004010CF 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:004010D3 04 09                    add     al, 9
text:004010D5 34 70                    xor     al, 70h
text:004010D7 88 44 24 61              mov     [esp+0A0h+encrypted_string_5], al
text:004010DB 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:004010DF 04 0A                    add     al, 0Ah
text:004010E1 34 65                    xor     al, 65h
text:004010E3 88 44 24 62              mov     [esp+0A0h+encrypted_string_5+1], al
text:004010E7 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:004010EB 04 0B                    add     al, 0Bh
text:004010ED 34 61                    xor     al, 61h
text:004010EF 88 44 24 63              mov     [esp+0A0h+encrypted_string_5+2], al
text:004010F3 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:004010F7 04 0C                    add     al, 0Ch
text:004010F9 34 72                    xor     al, 72h
text:004010FB 88 44 24 64              mov     [esp+0A0h+encrypted_string_5+3], al
text:004010FF 8B 44 24 54              mov     eax, [esp+0A0h+key]
text:00401103 04 0D                    add     al, 0Dh
text:00401105 C6 44 24 66 00           mov     [esp+0A0h+encrypted_string_5+5], 0
text:0040110A 34 73                    xor     al, 73h
text:0040110C 88 44 24 65              mov     [esp+0A0h+encrypted_string_5+4], al
text:00401110 8A 44 24 58              mov     al, [esp+0A0h+encrypted_string_8]
text:00401114 33 C0                    xor     eax, eax
```



```
.text:00401116
.text:00401116                         loc_401116:
.text:00401116 8A 4C 04 58             mov     cl, [esp+eax+0A0h+encrypted_string
.text:0040111A 8B 54 24 54             mov     edx, [esp+0A0h+key]
.text:0040111E 02 D0                   add     dl, al
.text:00401120 32 D1                   xor     dl, cl
```

*Figure 4: Compiled ADVobfuscator Exemplar Samples*

## BazarBackdoor/BazarLoader

BazarLoader and BazarBackdoor are malware families attributed to the TrickBot threat group, a.k.a. ITG23. Both are written in C++ and compiled for 64bit and 32bit Windows. BazarLoader is known to download and execute BazarBackdoor, and both use the Emercoin DNS domain (.bazar) when communicating with their C2 servers.

Other attributes of the loader and backdoor include extensive use of API function hashing and string obfuscation where each string is encrypted with varying keys. The string obfuscation methodology implemented in these files is interesting when compared with the ADVobfuscator samples previously described.

### Bazar string obfuscation

The string obfuscation implemented in variants of BazarLoader and BazarBackdoor is similar to what is implemented in *ADVobfuscator*. For example, the BazarBackdoor sample *189cbe03c6ce7bdb691f915a0ddd05e11adda0d8d83703c037276726f32dff56* detailed in Figure 5 contains a modified version of the string obfuscation techniques found in *ADVobfuscator*. In Figure 5, the string is moved to the stack four bytes at a time and the key used in the decryption loop is four bytes.

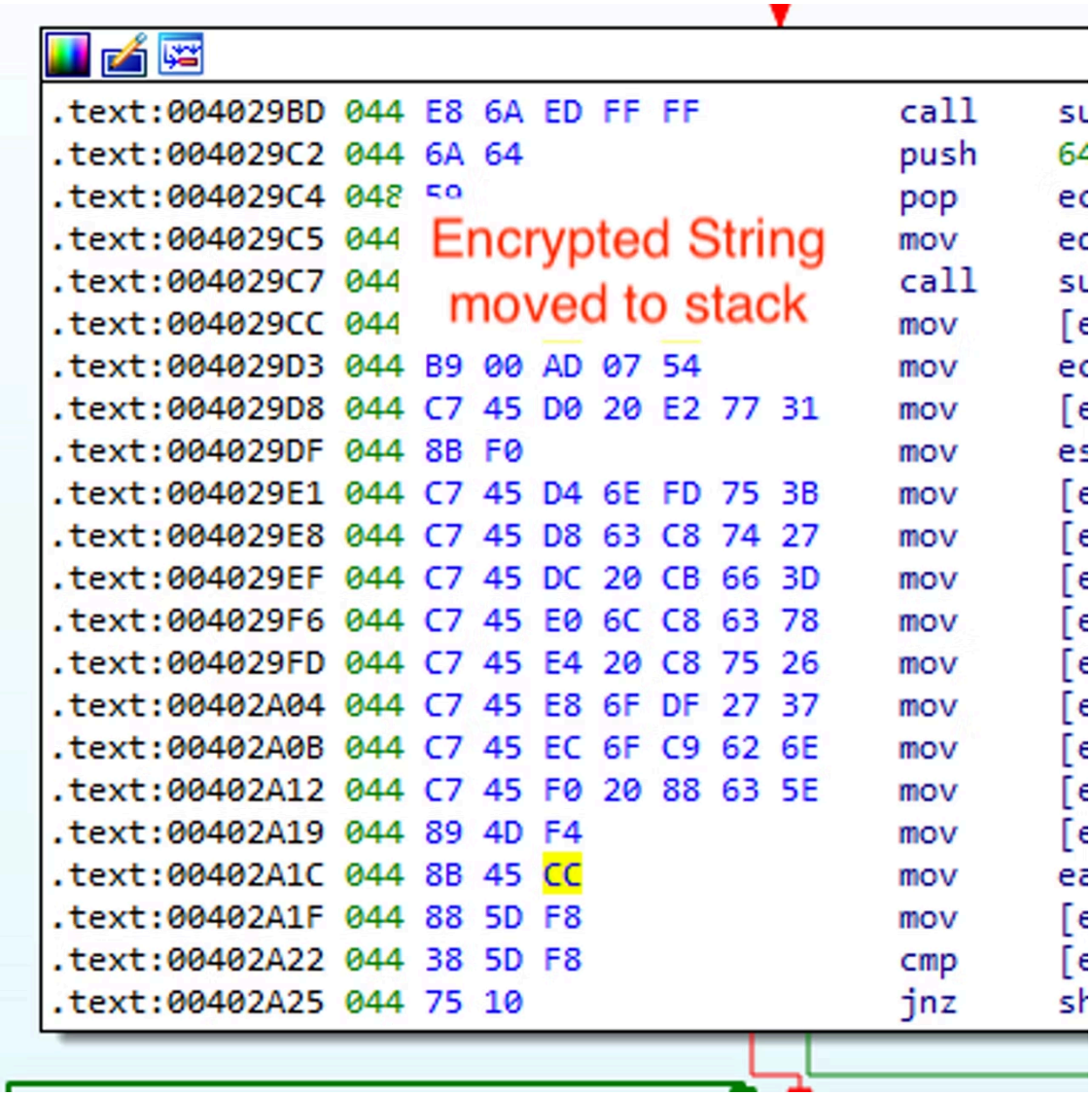



*Figure 5: XOR String Decryption 1*

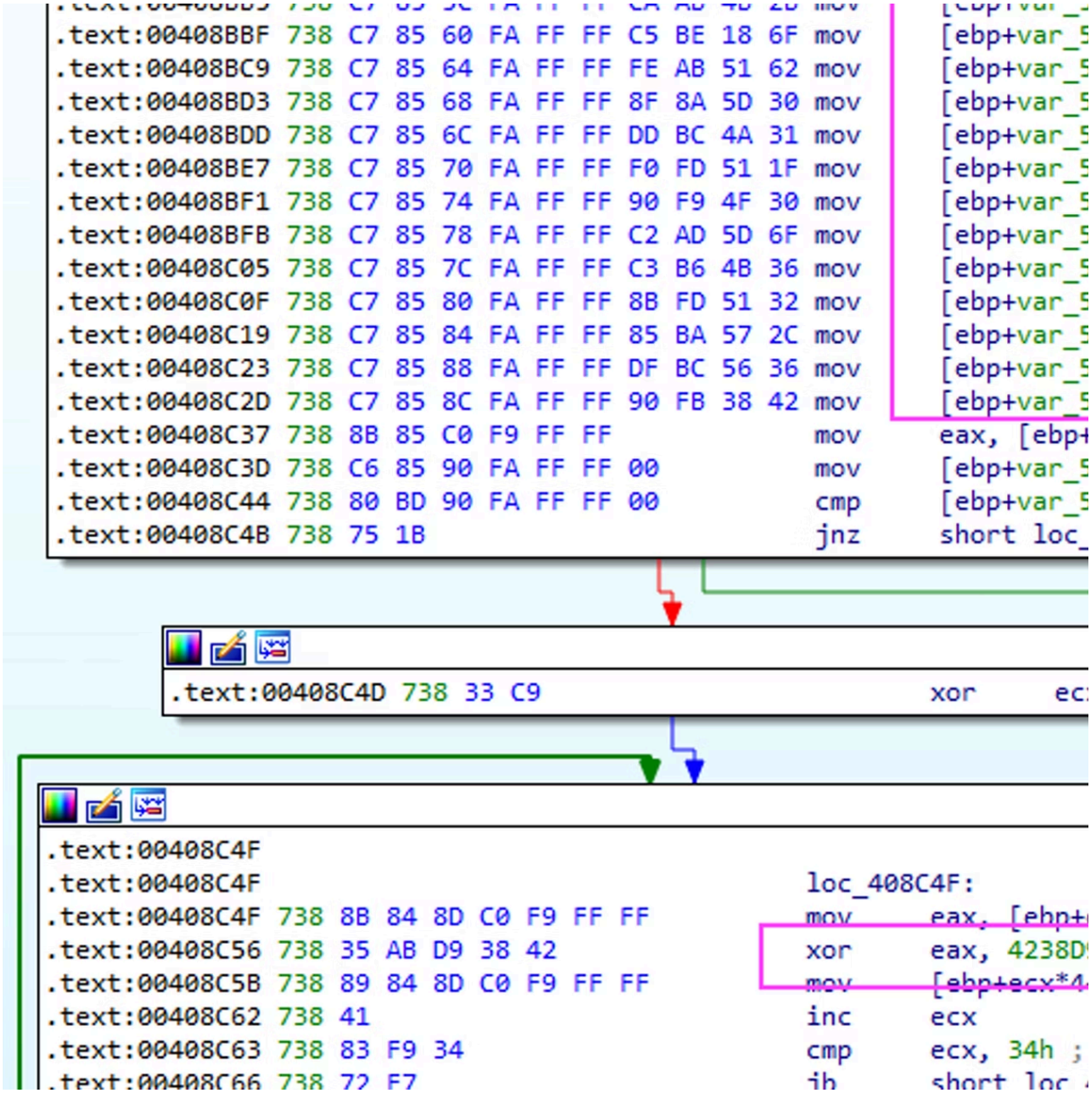



*Figure 6: XOR String Decryption 2*

## TrickBot and Bazar — Ongoing code evolution

Based on the similarities discovered through the analysis performed by X-Force, it is evident that the authors of BazarLoader and BazarBackdoor malware utilize template-based metaprogramming. While it is possible to break the resulting string obfuscation, the ultimate intent of the malware author is to hinder reverse engineering and evade signature-based detection. Metaprogramming is just one tool in the threat actors' toolbox. Understanding how these techniques work helps reverse engineers create tools to increase the efficiency of analysis and stay in step with the constant threat malware poses.

---

Source: https://securityintelligence.com/posts/trickbot-gang-template-based-metaprogramming-bazar-malware/